Pascal Pons Follow

SOLVING CONNECT FOUR

Part 10 – Better move ordering

Move exploration order is a key part of alpha-beta algorithm. It is most efficient when best moves are first explored. In part 5 we ordered the moves by exploring first the middle columns. It can be improved by taking into account the moves that are creating alignment opportunities.

Ordering moves with a score function

We will create a score function to order the moves to be explored. In case of tie with the score function we will continue to keep the previous ordering heuristic by exploring from central columns to side columns. Moves creating open 3-alignments generate winning opportunities that could be leveraged later in the game. These moves are likely to be, in average, better moves than others.

Our score function we will simply count the number of winning positions after playing a given possible move. Moves creating winning opportunities, will increase this count by one (or more).

Implementation

The score function is straitghforward using the compute_winning_position() function. Indeed, this function computes a bitmap of all winning spots of the current player. We only need to count the number of bits equal to one in this bitboard. This bit counting problem is known as population count.

We selected an efficient implementation of population count when the number of one is low, which is often the case for real connect four positions. This leads to the following implementation of a score function as part of the Position class:

```
    /**
     * Score a possible move.
     * @param move, a possible move given in a bitmap format.
     * The score we are using is the number of winning spots
     * the current player has after playing the move.
     */
    int moveScore(uint64_t move) const {
      return popcount(compute_winning_position(current_position | move, mask));
    }

    /**
     * counts number of bit set to one in a 64bits integer
     */
    static unsigned int popcount(uint64_t m) {
      unsigned int c = 0;
      for (c = 0; m; c++) m &= m - 1;
      return c;
    }
```

We also implement a simple sorter helper class to help sorting moves by score, while preserving initial insersion ordering in case of tie. We use the insertion sort algorithm that is stable sort algorithm and has quite efficient implementation for small arrays.

```
/*
 * This class helps sorting the next moves
 *
 * You have to add moves first with their score
 * then you can get them back in decreasing score
 *
 * This class implement an insertion sort that is in practice very
 * efficient for small number of move to sort (max is Position::WIDTH)
 * and also efficient if the move are pushed in approximatively increasing
 * order which can be acheived by using a simpler column ordering heuristic.
 */
class MoveSorter {
public:

  /*
   * Add a move in the container with its score.
   * You cannot add more than Position::WIDTH moves
   */
  void add(uint64_t move, int score)
  {
    int pos = size++;
    for(; pos && entries[pos-1].score > score; --pos) entries[pos] = entries[pos-1];
    entries[pos].move = move;
    entries[pos].score = score;
  }

  /*
   * Get next move
   * @return next remaining move with max score and remove it from the container.
   * If no more move is available return 0
   */
  uint64_t getNext()
  {
    if(size)
      return entries[--size].move;
    else
      return 0;
  }
private:
  // number of stored moves
  unsigned int size;

  // Contains size moves with their score ordered by score
  struct {uint64_t move; int score;} entries[Position::WIDTH];
};
```

Then the integration in negamax is quite simple, we compute the score of all possible next moves in the previous order (from central to side columns) and insert them in the move sorter helper. Then we iterate on them in the sorted order using the getNext() iterator.

Full source code corresponding to this part.

Benchmark

Improving the node exploration order allows to huge improvement in the pruning. The number of explored node had been divided by up to 100 in most complex cases. Meanwhile, the score computation has a quite expensive cost and multiplies by about 3 the average computational time per node.

Overall, it is of course a huge improvement for most complex positions, it now allows to solve the Hardest test set (Begin-Hard) in a reasonable time and faster than the Fhourstone benchmark.

Solver	Test Set name	mean time	mean nb pos	K pos/s
Better move ordering (strong solver)	End-Easy	5.109 µs	54.93	10,750
Better move ordering (strong solver)	Middle-Easy	42.74 µs	517.2	12,100
Better move ordering (strong solver)	Middle-Medium	4.108 ms	48,290	11,760
Better move ordering (strong solver)	Begin-Easy	296.4 µs	3,692	12,460
Better move ordering (strong solver)	Begin-Medium	126.2 ms	1,449,000	11,480
Better move ordering (strong solver)	Begin-Hard	6.454 s	71,440,000	11,070
Better move ordering (weak solver)	End-Easy	3.638 µs	31.46	8,649
Better move ordering (weak solver)	Middle-Easy	56.49 µs	618.4	10,950
Better move ordering (weak solver)	Middle-Medium	2.141 ms	23,810	11,120
Better move ordering (weak solver)	Begin-Easy	2.301 ms	25,480	11,070
Better move ordering (weak solver)	Begin-Medium	59.05 ms	663,100	11,230
Better move ordering (weak solver)	Begin-Hard	3.734 s	41,460,000	11,100

Tutorial plan

SOLVING CONNECT FOUR

Updated: October 22, 2017

Previous **Next**